Querying Proofs

David Aspinall[1], Ewen Denney[2], and Christoph Lüth[3]

[1] LFCS, School of Informatics
University of Edinburgh
Edinburgh EH8 9AB, Scotland
[2] SGT, NASA Ames Research Center
Moffett Field, CA 94035, USA
[3] Deutsches Forschungszentrum für Künstliche Intelligenz
Bremen, Germany

Abstract. We motivate and introduce a query language PrQL designed for inspecting machine representations of proofs. PrQL natively supports *hiproofs* which express proof structure using hierarchical nested labelled trees. The core language presented in this paper is *locally structured* (first-order), with queries built using recursion and patterns over proof structure and rule names. We define the syntax and semantics of locally structured queries, demonstrate their power, and sketch some implementation experiments.

1 Introduction

Automated proof tools and interactive theorem provers are increasingly called upon to produce evidence of their claims, in the form of representations of proofs that may be independently checked or, perhaps, imported into other systems. Proofs must connect together atomic rules of inference and axioms in a sound way according to an underlying logic. Checking that this has been done correctly is straightforward, although producing a proof in the first place may be extraordinarily difficult.

Real proofs can be very large, perhaps consisting of tens or hundreds of thousands of atomic rules of inference. There are many things that are interesting to know about such objects, beyond the basic fact that they are correctly constructed. For example, some natural questions when *inspecting* a proof are:

- What is the high-level structure of this proof, (how) can we break it down into pieces to understand it?
- Given a proof of a property which exploits a set of domain-specific axioms, which axioms actually occurred in the proof? (Or, in a purely logical setting, does a proof rely on axioms of classical logic?)
- Given a problem statement which contains some existential propositions as sub-formulae, which, if any, witnesses were found to make them true?
- Does a large proof contain duplicated parts that could be abstracted (or generalised) into a separate lemma, using a cut-like rule to reduce the size of the proof?

When the user is trying to understand the proof construction process, there are natural questions which relate the constructed proof back to the procedures that produced it. If tactics are our notion of proof producing procedure, some questions *relating* the proof to the tactics that produced it are:

– Given a set of tactics and a proof, which tactics were invoked in producing the proof and what subgoals did they solve?
– Were any tactics used repeatedly in this proof, perhaps with similar or identical inputs?
– Did some tactics get invoked but do no useful work?
– Given a failed proof (represented as a proof with unproved portions), which tactics were tried on the unproved portions?

These sort of questions are not idle curiosities: they are useful for practical *proof engineering*, when managing and maintaining sets of properties, proofs and programs which create and check them. One of us (Denney) routinely resorts to low-level scripted tools to perform these kind of examinations when building large safety cases supported by formal proofs.

We consider querying proofs here in a rigorous manner with the hope of enabling more general tools with clear foundations. In this paper, we introduce the basis of a query language PrQL designed specifically for querying proofs.

Hierarchical structured proofs. The foundation we start from is *Hiproofs* [1,2], which provide a simple abstract notion of proof tree by composing atomic rules of inference from an unspecified underlying logic. Going beyond ordinary trees, they have a notion of *hierarchy*, by allowing labelling and nesting subtrees. This simple addition provides a precise and useful notion of *structure* in the proof which can be used, for example, for noting where a lemma was applied, or where a particular tactic or external proof tool produced a subtree.

Contributions and paper outline. This paper contributes towards generic foundational aspects of theorem proving systems. Specifically, we design a proof query language from first principles, directly connected with a precise abstract notion of proof. With the help of some implementation experiements, we establish that it is useful. Although query languages for tree and graph structured data have been studied over the last decade or so, they have very rarely been applied to formal proofs.

The rest of this paper is structured as follows. Section 2 introduces the underlying setting of hiproofs used in the rest of the paper. Section 3 describes the design decisions we took for our query language, and introduces it with a sequence of informal examples and their intended meanings. Section 4 then describes the meaning of queries formally, and shows that example queries indeed have the denotations expected. In Section 5 we describe some experimental implementations of the query language. These are early ideas, presented to demonstrate some possible practical end points of our work rather than a complete implementation study. We give pointers and discussion of some related work in the concluding Section 6.

2 Hiproofs

Hiproofs add structure to an underlying *derivation system*, a simple form of logical framework. We give a brief recap here, for fuller details please see [2,1].

A hiproof is built from (inverted) atomic inference rules a in the underlying derivation system, to which we give a functional reading: a hiproof maps a finite list of input goals $[\gamma_1, \ldots, \gamma_n]$ to a list of output subgoals $[\gamma'_1, \ldots, \gamma'_m]$. Such a hiproof has the *arity* $n \rightarrow m$. A nested hiproof, appearing immediately inside a labelled box, has a single input goal which is the root of the tree at that level.

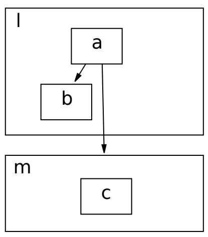
Informally and graphically, we draw hiproofs as inverted trees with a nested structure. Denotationally, a hiproof can be understood as a pair of an ordered tree and a forest with the same set of nodes, subject to some well-formedness conditions. Syntactically, a hiproof can be written as a term s in this grammar:

$$\begin{array}{rll} s ::= & a & \text{atomic} \\ \mid & \mathsf{id} & \text{identity} \\ \mid & [l]\, s & \text{labelling} \\ \mid & s_1 \;;\; s_2 & \text{sequencing} \\ \mid & s_1 \otimes s_2 & \text{tensor (juxtaposition)} \\ \mid & \langle\rangle & \text{empty} \end{array} \tag{1}$$

Fig. 1 shows an example hiproof term and its graphical representation in the middle. Boxes indicate nestings and have labels in their top corners; unlabelled boxes contain atomic rules. Tensor $\otimes$ places things side-by-side and sequencing ; builds "wiring" to connect things together, using identity to create wires where a goal is not manipulated. In the example, id exports the second subgoal from the atomic rule a outside the box labelled l. The empty proof $\langle\rangle$ is useful when building proofs programmatically.

Valid hiproofs. A hiproof is called *valid* if it corresponds to a real proof tree in the underlying derivation system. The hiproof term in Fig. 1 validates the proof tree shown on the right-hand side, where an input goal γ_1 is proved using the atomic inference rules a, b and c. Validity extends naturally to arbitrary hiproof terms that have more than one input goal; such a term corresponds to a finite sequence of proof trees. We write $s \;\vdash\; g_1 \longrightarrow g_2$ if s is valid in this more

$([l]\, a \;;\; b \otimes \mathsf{id}) \;;\; [m]\, c$



$$\dfrac{\dfrac{}{\gamma_2}\,\mathrm{b} \qquad \dfrac{}{\gamma_3}\,\mathrm{c}}{\gamma_1}\,\mathrm{a}$$

Fig. 1. A hiproof, its graphical representation and a proof it validates.

$$\frac{\frac{\gamma_1\cdots\gamma_n}{\gamma}a \text{ is an atomic inference}}{a \vdash \gamma \longrightarrow [\gamma_1,\ldots,\gamma_n]} \qquad \frac{}{\mathsf{id} \vdash \gamma \longrightarrow \gamma} \qquad \frac{s \vdash \gamma \longrightarrow g}{[l]\, s \vdash \gamma \longrightarrow g} \qquad \frac{}{\langle\rangle \vdash [\,] \longrightarrow [\,]}$$

$$\frac{s_1 \vdash g_1 \longrightarrow g \quad s_2 \vdash g \longrightarrow g_2}{s_1 \,;\, s_2 \vdash g_1 \longrightarrow g_2} \qquad \frac{s_1 \vdash g_1 \longrightarrow g_1' \quad s_2 \vdash g_2 \longrightarrow g_2'}{s_1 \otimes s_2 \vdash g_1 \,^\wedge g_2 \longrightarrow g_1' \,^\wedge g_2'}$$

Fig. 2. Validation of Hiproofs.

general sense, taking a list of input (proven) goals g_1 to produce a list of output (unsolved) goals g_2. This relation is defined by the rules in Fig. 2, where $^\wedge$ stands for list append.

Validity checking can be seen as a way of adding goals to a hiproof; correspondingly, a valid hiproof can be seen as a nested labelling applied to a flat proof. In this paper we restrict our attention to valid hiproofs and we assume that the goals are uniquely determined by (or implicit within) the validated hiproof.

3 Local Structured Queries

How should we express queries on proof such as those in Section 1? One design choice would be to take an existing query language for graph (or semi-structured) data models (e.g., see [3] for models and [4] for web query languages), and then map from hiproofs into the existing language and use queries there. The drawback with that approach is that we immediately lose connection with our particular source language, where it is most natural to express and study our queries. So instead we shall start from queries written in a minimal native query language, and investigate a direct semantics for them.

Our queries follow the hiproof structure, matching on leaves with atomics, structured proofs using labels, or on input or output goals of subproofs. We consider queries which specify structure only *locally*, in the sense that they cannot directly compare one part of the tree with another, or measure absolute position within the global proof. This restriction arises because we use only first-order variables that refer to names and goals, not to subtrees or paths. Despite this, the language is still rather expressive and captures most of our desired queries.

To introduce the language, we begin with constructs for matching leaves and goals within proofs, and then build up following the linear hiproof syntax.

Matches. We build *matches* inside queries using wildcards, variables (which may get instantiated), constants (which have to match exactly) and, in the case of goals, some goal filtering predicates ψ. Let Var_N be a set of schematic variables standing for names, ranged over by N in general and A when we suggest an atomic rule name or L a label name. Let Var_G be a set of variables standing for lists of goals. The name matches and goal matches are given by:

$$\begin{aligned} nm &::= a \mid l \mid * \mid N \\ gm &::= [\psi_1, \ldots, \psi_n] \mid G \end{aligned}$$

where ψ stands for a logic-dependent predicate on goals γ used to check some structural property of the goal term. For example we might have a predicate that checks whether a goal γ is in the form of a horn clause, when $\phi_{hornclause}(\gamma)$ holds. In [2] we used such predicates in a tactic **assert** ψ to influence the course of a proof, by failing if $\psi(\gamma)$ does not hold for the current goal. Most simply, we suppose that we always have a predicate to check for equality with any specific goal γ and we overload γ to stand for that predicate.

We use matches to build up the *basic queries* which specify local structure. Informally, a basic query may hold for a given hiproof and a substitution of variables the query contains; we will define the result of a query to be the set of variable instantiations that make it true. As (merely) a matter of style, we use a verbose SQL-like textual notation:

$q ::= *$	anything non-empty
$\mid$ **nothing**	nothing (matches only identity)
$\mid$ **atomic** nm	atomic rule match
$\mid$ **inside** $nm\ q$	q satisfied inside box with label matching
$\mid$ q_1 **then** q_2	q_1 and q_2 satisfied by successive nodes in ;
$\mid$ q_1 **beside** q_2	q_1 and q_2 satisfied by adjacent nodes in $\otimes$
$\mid$ **ingoals** gm	goals into sub-proof match
$\mid$ **outgoals** gm	goals out of sub-proof match

Notice that the subject of the query is left implicit, phrases act as anchored patterns. This core is almost the same language as the hiproof syntax itself, omitting empty proofs and adding the ability to match on goals within the proof. Queries built using the first five of these constructors are called *elementary*.

For the hiproof given in Fig. 1, the following queries are each satisfied (the alignment around **then** matches the vertical split):

$$\begin{aligned} (\textbf{inside}\ \mathsf{l}\ *)\ &\textbf{then}\ (\textbf{inside}\ \mathsf{m}\ *) \\ (\textbf{inside}\ *\ *\ \textbf{then}\ *\ \textbf{beside}\ \textbf{nothing})\ &\textbf{then}\ * \\ (\textbf{inside}\ L_1\ *)\ &\textbf{then}\ (\textbf{inside}\ *\ \textbf{atomic}\ A) \end{aligned}$$

The first two are purely structural, matching the form of the tree. The first matches the outer structure consisting of the box labelled l followed by the box labelled m. The second examines the shape inside the first box. The final query is satisfiable with the (unique) instantiation $L_1 \mapsto \mathsf{l}, A \mapsto \mathsf{c}$.

Connectives. We allow propositional logical connectives to build compound queries, with familiar intended meanings:

$$\begin{aligned} q ::= &\ \ldots \\ \mid &\ q_1 \wedge q_2 \\ \mid &\ q_1 \vee q_2 \\ \mid &\ \neg q \end{aligned}$$

Search and check. Two important quantifier combinators on queries allow us to search within a proof for somewhere that a query is satisfied, or check that a query is satisfied everywhere. With a syntactic interpretation, the natural domain of quantification is by subterm; because any subterm of a valid hiproof is also valid, this makes sense and we equate "subproof" (in the hiproof sense) with subterm.

$$\begin{array}{lll} q ::= \ldots & & \\ \quad \mid \ \textbf{somewhere}\ q & & q \text{ holds in some subproof} \\ \quad \mid \ \textbf{everywhere}\ q & & q \text{ holds in every subproof} \end{array}$$

The scope of **somewhere** and **everywhere** extends as far right as possible. These queries might be added directly to the language meaning, but we define them instead using recursion (introduced below). The **somewhere** combinator is used in many of our examples. For example, a proof uses a tactic tac if the query

$$\textbf{somewhere inside}\ \mathsf{tac}\ *$$

is satisfied. As another example, we use a match on a goal-list variable G to find the goals passed into a tactic. The query

$$(\textbf{somewhere inside}\ \mathsf{m}\ \textbf{ingoals}\ G) \vee (\textbf{somewhere atomic}\ \mathsf{b} \wedge \textbf{ingoals}\ G)$$

can be read as "tell me the goals that are input to tactic m or the atomic rule b". This would return the pair of instantiations $\{G \mapsto [\gamma_2], G \mapsto [\gamma_3]\}$ for the hiproof in Fig. 1.

When is **everywhere** useful? Clearly not for anything that requires a fixed structure, but with a goal-matching assertion that checks the format of the goals, for example, the check

$$\textbf{everywhere outgoals}\ [\phi_{hornclause}]$$

requires that every goal appearing in the tree must have that certain form. With conditional queries we can specify that only goals appearing in certain places must have some property.

Recursive queries. Just as with tactics we can allow recursively defined queries. Recursively defined queries allow us to build up regular patterns. Singly-recursive queries can be defined using query variables Q:

$$\begin{array}{l} q ::= \ldots \\ \quad \mid \ \mu Q.q \end{array}$$

where q is a query in which Q can appear free. An example of a pattern query is:

$$\mu Q.\,(\textbf{atomic}\ \mathsf{a}\ \textbf{then}\ (\textbf{ingoals}\ [\gamma_2]\ \textbf{beside}\ Q)) \vee (\textbf{inside}\ \mathsf{m}\ *)$$

which is satisfied by proofs that repeatedly apply the atomic rule a, until reaching a box named m.

Using recursion we can define the searching and checking quantifiers described above:

$$\begin{aligned}\textbf{somewhere}\ q \overset{def}{=} \mu Q.\, q \vee\ & (\textbf{inside}\ *\ Q) \vee (Q\ \textbf{beside}\ *) \vee (*\ \textbf{beside}\ Q) \vee \\ & (Q\ \textbf{then}\ *) \vee (*\ \textbf{then}\ Q) \\ \textbf{everywhere}\ q \overset{def}{=} \mu Q.\, q \wedge\ & ((\textbf{inside}\ *\ Q) \vee (Q\ \textbf{beside}\ Q) \vee \\ & (Q\ \textbf{then}\ Q) \vee \textbf{nothing} \vee \textbf{atomic}\ *))\end{aligned}$$

these ensure that q holds at one (or every) node following the structure of the proof: notice that exactly one of the disjuncts must hold in the recursive cases. Later on we will show that these definitions have the intended meaning.

Derived forms. Using this core, we can readily add more derived forms:

$$\begin{aligned}q_1\ \textbf{when}\ q_2 &\overset{def}{=} \neg q_2 \vee q_1 \\ \textbf{provesgoal}\ \gamma &\overset{def}{=} \textbf{ingoals}\ [\gamma] \wedge \textbf{outgoals}\ [] \\ \textbf{axiom}\ nm &\overset{def}{=} \textbf{atomic}\ nm \wedge \textbf{outgoals}\ [] \\ \textbf{islabel}\ nm &\overset{def}{=} \textbf{inside}\ nm\ * \\ \textbf{isthen} &\overset{def}{=} *\ \textbf{then}\ * \\ \textbf{whenin}\ nm\ q &\overset{def}{=} \textbf{inside}\ nm\ q\ \textbf{when islabel}\ nm \\ \textbf{somewherealong}\ q &\overset{def}{=} \mu Q.\, q \vee (Q\ \textbf{beside}\ *) \vee (*\ \textbf{beside}\ Q) \\ \textbf{separately}\ q_1\ q_2 &\overset{def}{=} \mu Q.\, (\textbf{somewhere}\ q_1\ \textbf{beside somewhere}\ q_2) \\ &\qquad \vee (\textbf{somewhere}\ q_1\ \textbf{then somewhere}\ q_2) \\ &\qquad \vee (\textbf{inside}\ *\ Q) \\ \textbf{nearby}\ q &\overset{def}{=} \mu Q.\ q \vee (Q\ \textbf{beside}\ *) \vee (*\ \textbf{beside}\ Q) \\ &\qquad \vee (Q\ \textbf{then}\ *) \vee (*\ \textbf{then}\ Q)\end{aligned}$$

The **when** conditional combinator is satisfied if q_1 is satisfied whenever q_2 is; by convention, the scope of q_1 and q_2 in **when** extend as far as possible. The last three combinators again use recursion to expand the scope of the local structure specifications. The query **somewherealong** q is satisfied if q is satisfied in a $\otimes$-list of hiproofs; **separately** q_1 q_2 requires that q_1 and q_2 hold on disjoint portions of the proof, arbitrarily separated. The query **nearby** q is an adjusted version of **somewhere** which restricts to the same level (without descending into labelled boxes).

3.1 Examples

Now we can write many of our motivating examples. For example, the input goals to tactic `tac` are simply the G that satisfy:

$$\textbf{somewhere inside}\ \mathsf{tac}\ \textbf{ingoals}\ G.$$

The tactic tac occurs recursively in a hiproof if the query

$$\textbf{somewhere inside } \mathsf{tac} \textbf{ somewhere islabel } \mathsf{tac}$$

is satisfied. The tactic inner always occurs whenever the tactic outer is invoked:

$$\textbf{everywhere whenin } \mathsf{outer} \textbf{ somewhere islabel } \mathsf{inner}.$$

A tactic named base always appears alongside a tactic named step inside the tactic induct:

$$\begin{aligned}\textbf{everywhere whenin } \mathsf{induct} \textbf{ somewhere } &(\textbf{somewherealong islabel } \mathsf{base})\\ &\wedge (\textbf{somewherealong islabel } \mathsf{step}).\end{aligned}$$

Further examples are given after looking at the semantics.

4 Semantics

We will define the semantics of queries using a satisfication relation $s \models_\sigma q$. This denotes satisfication of a structured query on a hiproof s with respect to a substitution σ for match variables. The substitution maps variables N to names for atomic tactics and labels, and variables G to lists of the form $[\gamma_1, \ldots, \gamma_n]$.

Two base satisfaction relations define matching on names and goal lists:

$$\begin{array}{rlll} * & \models_\sigma n & & \text{always} \\ n' & \models_\sigma n & \text{iff} & n = n' \\ N & \models_\sigma n & \text{iff} & \sigma(N) = n \end{array}$$

$$\begin{array}{rlll} [\psi_1, \ldots, \psi_n] & \models_\sigma g & \text{iff} & \exists \gamma_1 \cdots \gamma_n .\, g = [\gamma_1, \ldots, \gamma_n] \text{ and } \psi_1(\gamma_1) \cdots \psi_n(\gamma_n) \\ G & \models_\sigma g & \text{iff} & \sigma(G) = g \end{array}$$

Before giving the main relation, we consider hiproof terms in more detail. Terms s in the hiproof grammar denote tree-based models in the denotational semantics of hiproofs [1]. Under the denotational interpretation, certain terms are equivalent. We will give our interpretation over the syntax, but closing under this equivalence. More specifically, we consider valid hiproofs given in Sect. 2 modulo these equations:

$$\begin{array}{rlr} s \,;\, \mathsf{id} &= s & \mathsf{id} \text{ is an identity for sequencing} \\ \mathsf{id} \,;\, s &= s & \\ s \otimes \langle\rangle &= s & \langle\rangle \text{ is an identity for juxtaposition} \\ \langle\rangle \otimes s &= s & \\ s \,;\, \langle\rangle &= s & \langle\rangle \text{ is a right-identity for sequencing} \\ s_1 \,;\, (s_2 \,;\, s_3) &= (s_1 \,;\, s_2) \,;\, s_3 & ; \text{ is associative} \\ s_1 \otimes (s_2 \otimes s_3) &= (s_1 \otimes s_2) \otimes s_3 & \otimes \text{ is associative} \\ (s_1 \,;\, s_2) \otimes (s_3 \,;\, s_4) &= (s_1 \otimes s_3) \,;\, (s_2 \otimes s_4) & ; \text{ and } \otimes \text{ can be exchanged} \end{array}$$

These equations are justified by the denotational semantics (which we omit from this paper), and it is easy to confirm that the equations preserve validity on the same lists of input and output goals for the rules in Fig. 2. We will write $s = s'$ if two terms are equal in the theory generated by these equations (i.e., closing also under congruence). We reserve $s \equiv s'$ to denote syntactic identity.

Definition 1 (Query satisfaction). *Let s be a valid hiproof and q a query in the minimal query language. The satisfaction of q for s with the substitution σ is defined as the least relation $s \models_\sigma q$ satisfying:*

$s \models_\sigma *$	*when*	$s \neq \langle\rangle$
$\mathsf{id} \models_\sigma$ **nothing**		
$a \models_\sigma$ **atomic** nm	*when*	$nm \models_\sigma a$
$[l]\, s \models_\sigma$ **inside** $nm\ q$	*when*	$nm \models_\sigma l$ *and* $s \models_\sigma q$
$s_1 ; s_2 \models_\sigma q_1$ **then** q_2	*when*	$s_1 \models_\sigma q_1$ *and* $s_2 \models_\sigma q_2$
$s_1 \otimes s_2 \models_\sigma q_1$ **beside** q_2	*when*	$s_1 \models_\sigma q_1$ *and* $s_2 \models_\sigma q_2$
$s \models_\sigma$ **ingoals** gm	*when*	$gm \models_\sigma g$ *where* $s \vdash g \longrightarrow h$
$s \models_\sigma$ **outgoals** gm	*when*	$gm \models_\sigma h$ *where* $s \vdash g \longrightarrow h$
$s \models_\sigma q_1 \wedge q_2$	*when*	$s \models_\sigma q_1$ *and* $s \models_\sigma q_2$
$s \models_\sigma q_1 \vee q_2$	*when*	$s \models_\sigma q_1$ *or* $s \models_\sigma q_2$
$s \models_\sigma \neg q$	*when*	$\neg(s \models_\sigma q)$
$s \models_\sigma \mu Q.q$	*when*	$s \models_\sigma q[\mu Q.q/Q]$
$s \models_\sigma q$	*when*	$\exists s'.\, s' \models_\sigma q$ *and* $s' = s$.

Recursive queries $\mu Q.q$ are interpreted using unfolding; this suffices since we query only finitely deep trees. More precisely, we can define satisfaction using an auxiliary relation $\models_n$ indexed by the maximum depth of the number of unfoldings of a recursive query, where $\mu_n Q.q$ can be unfolded at most n times. Then $\models$ is defined as the union of all finite unfolding relations $\models_n$. The definition works for singly recursive queries where we do not need to interpret queries with free query variables, but can be extended standardly for mutually recursive queries.

Proposition 1. *Let s be a valid hiproof. Then*

1. $s \models_\sigma$ **somewhere** q *iff* $\exists s'.s'$ *is a subterm of* s *and* $s' \models_\sigma q$,
2. $s \models_\sigma$ **everywhere** q *iff* $\forall s'.s'$ *is a subterm of* s *and* $s' \models_\sigma q$.

(where quantification ranges over non-empty terms, and s is a subterm of itself).

Thus these important derived forms have the intended meanings.

How precise are our queries? The following proposition establishes, as intended, that every term can be characterised up to equality by a query. Thus, we can use queries to describe finite sets of hiproofs.

Proposition 2. *Given any empty-normal hiproof s, there is a query $Q(s)$ which characterises s precisely.*

Proof. Let $Q(s)$ be given by the embedding:

$$\begin{aligned} Q(\mathsf{id}) &= \textbf{nothing} \\ Q([l]\,s) &= \textbf{inside}\ l\ Q(s) \\ Q(s_1\ ;\ s_2) &= Q(s_1)\ \textbf{then}\ Q(s_2) \\ Q(s_1 \otimes s_2) &= Q(s_1)\ \textbf{beside}\ Q(s_2) \\ Q(\langle\rangle) &= \neg * \end{aligned}$$

Now we claim that whenever $s' \models_\sigma Q(s)$ for some s', we must have $s = s'$.

How expressive are our queries? This is another natural question, which we discuss further below in Sec 4.3.

4.1 Examples and their results

Now we demonstrate how some of our motivating queries are written in our language; meanings can be calculated using the semantics above to show that they are as desired.

The invocation of a query to get some results can be written in SQL style as:

$$\textbf{select}\ e\ \textbf{from}\ s\ \textbf{where}\ q$$

which denotes the set of expressions $\sigma(e)$ for all substitutions σ that satisfy the query. That is:

$$\{\sigma(e) \mid s \models_\sigma q\}. \tag{2}$$

The kind of expressions e chosen here depends on what we want to do with query results. We don't consider a general transformation language for query results here, but one could easily allow expressions that combine pieces of query results in arbitrary ways, for example, building new hiproofs again over the query variables. Our examples below restrict to simple query variables.

- To find all the axioms in a valid hiproof s:

  $$\begin{aligned} \mathit{Axioms}(s) = {} & \textbf{select}\ A\ \textbf{from}\ s\ \textbf{where} \\ & \quad \textbf{somewhere axiom}\ A \end{aligned}$$

  Applied to $s = ([l]\,a\ ;\ b \otimes \mathsf{id})\ ;\ [m]\,c$, this query returns $\{A \mapsto \mathsf{c}, A \mapsto \mathsf{b}\}$.
- To find the existential witnesses inside a valid hiproof s, we can find uses of the existential introduction rule:

  $$\begin{aligned} \mathit{Wit}(s) = {} & \textbf{select}\ A\ \textbf{from}\ s\ \textbf{where} \\ & \quad \textbf{somewhere atomic}\ A \wedge \textbf{atomic}\ \mathit{ExI}_t \end{aligned}$$

  Strictly, this requires a slight generalisation of our atomic name matching to admit sets of names; we assume the *ExI* rule is annotated by the witness t that is chosen.

– Which goals are input to (or output from) a tactic called tac?

$$\begin{aligned} \mathit{Input}(\mathtt{tac}, s) &= \textbf{select } G \textbf{ from } s \textbf{ where} \\ &\qquad \textbf{somewhere inside } \mathtt{tac} \textbf{ ingoals } G \\ \mathit{Output}(\mathtt{tac}, s) &= \textbf{select } G \textbf{ from } s \textbf{ where} \\ &\qquad \textbf{somewhere inside } \mathtt{tac} \textbf{ outgoals } G \end{aligned}$$

– Which tactics call themselves recursively? (shown earlier for fixed tac)

$$\begin{aligned} \mathit{Rec}(s) = {} & \textbf{select } L \textbf{ from } s \textbf{ where} \\ & \qquad \textbf{somewhere inside } L \textbf{ somewhere islabel } L \end{aligned}$$

– Which tactic uses atomic tactic a, i.e., inside which label does a occur? Using the **nearby** combinator defined in the last section, this query returns all labels L which contain a directly, i.e., labels which are the immediate surrounding parent of a, not a more distant ancestor.

$$\begin{aligned} \mathit{Inside}(a, s) = {} & \textbf{select } L \textbf{ from } s \textbf{ where} \\ & \qquad \textbf{somewhere inside } L \textbf{ nearby atomic } a \end{aligned}$$

– Are there steps in the proof which have no effect?

$$\begin{aligned} \mathit{UselessTacs}(s) = {} & \textbf{select } L \textbf{ from } s \textbf{ where} \\ & \qquad \textbf{somewhere inside } L \textbf{ ingoals } G \wedge \textbf{outgoals } G \end{aligned}$$

This returns useless tactics that return the same goal that they were given (necessarily G must be a single element list by the hiproof structure). Of course, some tactics may be even worse and return the same goal that they were given and some more besides! To deal with these, we would need to extend goal matching to allow subset inclusion.

– Are there duplicated subproofs inside a proof? A good way to answer this is to look for subtrees that have the same input and output goal lists, using the **separately** operator introduced earlier:

$$\mathit{Duplicates}(s) = \textbf{select } G_i, G_o \textbf{ from } s \textbf{ where separately } q\ q$$

$$\text{where } q \text{ abbreviates } \textbf{ingoals } G_i \wedge \textbf{outgoals } G_o$$

Of course, to return (or locate) the actual subtrees that prove the same things, we would need to extend the query language with variables ranging over hiproofs (or paths in hiproofs). See Sect. 4.3 for discussion of this.

4.2 Query equivalence

Prop. 2 characterises proofs by queries. We can turn this around, and ask whether queries can be characterised by the proofs that satisfy them. This motivates a Leibniz-style equality between queries — two queries are equal if they are satisfied by the same proofs and the same substutions. This question is also motivated by complexity considerations — there may be more than one query to get a particular answer, but which one is more efficient?

Definition 2. *We say two queries q, p are* equivalent*, written $p \cong q$, if for all proofs s and substitutions σ, we have $s \models_\sigma q \Longleftrightarrow s \models_\sigma p$.*

With this definition, we have the following groups of equations. First, the logical connectives distribute over the basic queries:

$$\textbf{inside}\ nm\ (p \wedge q) \cong \textbf{inside}\ nm\ p \wedge \textbf{inside}\ nm\ p \tag{3}$$
$$\textbf{inside}\ nm\ (p \vee q) \cong \textbf{inside}\ nm\ p \vee \textbf{inside}\ nm\ p \tag{4}$$
$$p\ \textbf{then}\ (q_1 \wedge q_2) \cong (p\ \textbf{then}\ q_1) \wedge (p\ \textbf{then}\ q_2) \tag{5}$$
$$(p_1 \wedge p_2)\ \textbf{then}\ q \cong (p_1\ \textbf{then}\ q) \wedge (p_2\ \textbf{then}\ q) \tag{6}$$
$$p\ \textbf{then}\ (q_1 \vee q_2) \cong (q\ \textbf{then}\ q_1) \vee (p\ \textbf{then}\ q_2) \tag{7}$$
$$(p_1 \vee p_2)\ \textbf{then}\ q \cong (p_1\ \textbf{then}\ q \vee (p\ \textbf{then}\ q_2) \tag{8}$$
$$p\ \textbf{beside}\ (q_1 \wedge q_2) \cong (p\ \textbf{beside}\ q_1) \wedge (p\ \textbf{beside}\ q_2) \tag{9}$$
$$(p_1 \wedge p_2)\ \textbf{beside}\ q \cong (p_1\ \textbf{beside}\ q) \wedge (p_2\ \textbf{beside}\ q) \tag{10}$$
$$p\ \textbf{beside}\ (q_1 \vee q_2) \cong (p\ \textbf{beside}\ q_1) \vee (p\ \textbf{beside}\ q_2) \tag{11}$$
$$(p_1 \vee p_2)\ \textbf{beside}\ q \cong (p_1\ \textbf{beside}\ q \vee (p\ \textbf{beside}\ q_2) \tag{12}$$

Alternatively:

$$\textbf{inside}\ nm\ (p\Box q) \cong (\textbf{inside}\ nm\ p)\Box(\textbf{inside}\ nm\ pq) \tag{13}$$
$$(p_1\Box p_2)\otimes q \cong (p_1\otimes q)\Box(p_2\otimes q) \tag{14}$$
$$p\otimes(q_1\Box q_2) \cong (p\otimes q_1)\Box(p\otimes q_2) \tag{15}$$

for $\Box \in \{\wedge, \vee\}$ and $\otimes \in \{$ **then** , **beside** $\}$.

Secondly, negation distributes over the basic queries in the following fashion. For example, a query **inside** $lm\ q$ is not satisfied by s if s is either not of the form $[l]\, s'$, or if it is and s' does not satisfy q. This gives us:

$$\neg(\textbf{inside}\ lm\ q) \cong \textbf{inside}\ lm\ (\neg q) \vee (\textbf{atomic}\ *) \vee (*\ \textbf{beside}\ *) \vee (*\ \textbf{then}\ *) \tag{16}$$
$$\neg(p\ \textbf{beside}\ q) \cong (\neg p)\ \textbf{beside}\ * \vee *\ \textbf{beside}\ (\neg q) \vee (\textbf{atomic}\ *) \vee (\textbf{inside}\ *\ *) \vee (*\ \textbf{then}\ *) \tag{17}$$
$$\neg(p\ \textbf{then}\ q) \cong (\neg p)\ \textbf{then}\ * \vee *\ \textbf{then}\ (\neg q) \vee (\textbf{atomic}\ *) \vee (\textbf{inside}\ *\ *) \vee (*\ \textbf{beside}\ *) \tag{18}$$

Finally, we have the deMorgan equalities, double negation, commutativity and distributivity of conjunction over disjunction:

$$\neg(p \wedge q) \cong (\neg p) \vee (\neg q) \tag{19}$$
$$\neg(p \vee q) \cong (\neg p) \wedge (\neg q) \tag{20}$$
$$\neg(\neg p) \cong p \tag{21}$$
$$(p_1 \vee p_2) \wedge q \cong (p_1 \wedge q) \vee (p_2 \wedge q) \tag{22}$$
$$(p \wedge (q_1 \vee q_2) \cong (p \wedge q_1) \vee (p \wedge q_2) \tag{23}$$
$$\tag{24}$$

Equations (3) to (23) are proven by expanding Def. 2 and using Def. 1. **Example here? Seems all rather obvious.** From these equations, we can get the notion of a normal form.

Definition 3 (EEF and DNF). *A query is in* extended elementary form *(EEF), if it is an elementary query, a negation of an elementary query, or* **inside** *nm q where q is an extended elementary query.*

A query q is in disjunctive normal form (DNF), if it is of the shape $\bigvee_{i=1\dots n} \bigwedge_{j=1\dots m_i} \phi_{i,j}$ *where* $\phi_{i,j}$ *are extended elementary queries, or in other words a disjunction of conjunctions of extended elementary queries.*

Theorem 1. *For each query* q*, there is a an equivalent query* q'*, denotated as* DNF(q)*, such that* $q \cong q$ *and* q' *is in DNF.*

Proof. The proof proceeds by structural induction on q. Given an arbitrary q, we first push negation inside using equations (16) to (20). Now, using equations (3) to (12) we can push the query constructors (**inside**, **then**, **beside**) inside the logical connectives. Finally, use (22) and (23) to transform the formula into a disjunction of conjunctions.

ToDo:

– How big does it get? Blowup seems quadratic (note the negation equations only add a constant overhead) at each step– is that polynomial or exponential in total?
– Need to handle recursion.

4.3 Towards globally structured queries

There is a limit to what queries in our language can express. We call these queries *locally structured* because they are based on building up patterns of structure that are matched implicitly to a position in the tree. Using variable substitution and structural recursion, queries can span and relate different portions of the tree, but it is not possible to write a query that directly refers to (or returns) a position in the tree, or does any counting.

This limitation can be lifted, e.g., by adding a notion of *path* to the language, so we have a way to refer to positions in the tree. We have refrained from doing that in this paper to study the restricted case in detail first.

– TODO: something on expressivity/complexity: maybe say/show that with paths, queries are more succinct; maybe show a query that cannot be expressed although result is in the language (i.e., some substitution of labels/axioms/goals).

5 Implementing Queries

We have implemeted a prototype of the query language for small experiments. It represents queries as an algebraic datatype `Q`, and in time-honoured fashion uses SML as both implementation platform and scriptable command-line interface. Hiproofs are represented modulo the equations in Sect. 4, following the denotational semantics in [1]. The implementation is a functor which is generic over the proofs in question, reflecting the generic nature of the query language.

Corresponding to the semantic interpretation in Def. 1, the implementation provides a function:

```
val sat : P.HiPrf -> Q -> Subst list
```

which calculates the set of possible substitutions of query variables for names and goal lists to satisfy the given query; a satisfiable query with no substitution yields a singleton result of the empty substitution, while an unsatisfiable query yields the empty list as a result.

The function `sat` is a straightforward recursion over the structure of the query; the hiproof equations are taken care of by using a canonical (denotational) representation. For the basic queries an auxiliary function `match_nm` implements matching on atomic tactics and labels, and `match_gm` similarly for goals. Crucially, most compound queries are conjunctive: they decompose the argument, obtain sets of results for each of the subqueries, and then combine by pointwise unification of the resulting substitutions. (The unification here is very simple, just checking that the same variable is not mapped to different atoms, labels or goals, so it is a partial function.) In Haskell-like syntax, this is written as:

```
r1 >> r1 = [ s |  s1<-r1, s2<-r2, SOME s<-unify(s1, s2) ]
```

In contrast, to combine the results of disjunction it suffices to concatenate the result lists. Roughly, `sat` is implemented with cases like this:

```
sat (Atom a) (Atomic am) = match_nm(a, am)
sat (Lab l s) (Inside lm q)  = match_nm(l, lm) >> sat s q
sat (Seq s1 s2) (q1 Then q2) = sat s1 q1 >> sat s2 q2
...
sat s (q1 And q2) = sat s q1 >> sat s q2
sat s (q1 Or q2)  = sat s1 q1 @ sat s2 q2
sat s q = []
```

We provide two instantiations of the generic implementation: one for the syntactic hiproofs, where we have a datatype `S` as in (1), and one which models Isabelle proof objects as hiproofs.

5.1 Isabelle Proofs

Needs to be cleaned up. Notes about Isabelle proof objects:

- They have an inherent hierarchical structure, which may or may not be what we want. Boxes correspond to proven theorems; skeleton reduces a proof to the proof down to the axioms of the logic.

– How to model sequence and tensor? Need to divine from structure of proof and arity of rules (theorems).
– This means queries are not logic-independent. The axiom query always work, but the recursive-tactic query now means does a theorem use itself in its own proo.
– Isabelle proof objects do not have any information about the high-level proof scripts used to create them. We can add this information by using vacous "label theorems" which are just $\phi \longrightarrow \phi$, but the names of which carry labelling information. The hierarchical structure will now only take account of the label theorems.

5.2 TSTP

Proofs in TSTP consist of the sequence of formulas output by an automated theorem prover, including axioms, conjecture, and derived formulas. All formulas have a source. For derived formulas the source indicates the inference rule and antecedent formulas from which it was derived. For a leaf formula the source is the original location of the formula. TSTP, itself, does not have any notion of inference rule—these are supplied by the underlying provers.

Consider the underlying proof on the right of Fig. 1. This can be translated into TSTP in either a forwards style, where we start with formulas, γ_2 and γ_3 given by axioms b and c, respectively, and then apply inference rule, a, to conclude γ_1, or in a backwards style, where we start with conjecture γ_1, apply a to get two sub-goals, and then discharge them with the corresponding axioms. The TSTP representations of these derivations are quite different, though they have a single analogous proof tree.

Although TSTP does not represent tactics, inference rules can be nested, giving a simple form of hierarchy. We could also look to decompose the derivations thus deriving an implicit hierarchy, or extend the language with labels on sub-derivations to represent hierarchy explicitly.

6 Related work and conclusions

This paper introduced locally structured proof queries in our proof query language, PrQL. Much remains to be done. Further work will extend the language to globally structured (higher-order) queries and queries defined directly over our semantic models. To explain implementation strategies, we ought to give a more precise account of how queries are evaluated: this might be with a direct operational interpretation, or via an auxiliary mechanism. Further out, we want to set this work in the context of related query languages, perhaps by translations. We plan to investigate mappings from PrQL to both PML and TSTP. Via mappings to other foundational query languages for graph models, we may derive expressivity and complexity results (see e.g., [5]).

Related work in theorem proving. The idea of a general query language for inspecting formal proofs appears novel, although there are many investigations into exploiting proofs using ad hoc features to reference or manipulate parts of proofs. We can't survey all but mention a few. Connecting decision procedures to theorem proving, researchers added invocation records (and perhaps justifications) grafted into an overall proof or justification (e.g., [6]). Noteworthy sub-trees may be represented using names for reference (and then shared to create a dag structure) as in TPTP and its proof format TSTP [7]. Many systems use debugging output for proof procedures to create a lengthy log, which explains where things were tried and failed. Some tools use representations of proof trees in the first place which connect the proof-producing mechanism to the proof and are equipped with browsing and editing mechanisms, e.g., NuPrl [8]. The strand of research into *proof-carrying code* has taken the independence and dependability of proofs (and more generally, *certificates* for search procedures) very seriously [9]. Besides checking proofs, other researchers made efforts to *translate proofs* between systems [10]; ways to *discover dependencies* between parts of proofs [11] to help simplify or rearrange; and ways to *mine proofs* to discover common patterns [12].

TSTP proofs can be queried by translation [13] into the Proof Markup Language (PML) [14], which provides an interlingua representation for the justification of results produced by Semantic Web services. PML represents both proof and provenance level information. Queries in PML are simply partial proofs, rather than expressions in a separate query language (although PrQL also has similarities to its underlying proof language), and query evaluation seeks to return (possibly partial) proofs that "fill in the blanks" in the initial query. A browser also allows certain forms of querying.

Query languages for structured data and programs. Away from theorem proving, query languages for trees and graphs have been studied for some time. Languages related to PrQL include those aimed at semi-structured (XML-like) models such as UnQL [15] which uses structural recursion on tree (and graph) representations, similarly to PrQL's recursive queries, and Graph Logic [16] which uses a separating conjunction to destruct the graph subject of queries. Checking for patterns in programs, ASTLog [17] is a Prolog variant for examining syntax trees and PQL [18] is a more general framework for querying programs based on examining a program text at varying levels of abstraction.

References

1. Denney, E., Power, J., Tourlas, K.: Hiproofs: A hierarchical notion of proof tree. Electr. Notes Theor. Comput. Sci. **155** (2006) 341–359
2. Aspinall, D., Denney, E., Lüth, C.: Tactics for hierarchical proof. Mathematics in Computer Science **3**(3) (2010) 309–330

3. Angles, R., Gutierrez, C.: Survey of graph database models. ACM Comput. Surv. **40** (February 2008)
4. Bailey, J., Bry, F., Furche, T., Schaffert, S.: Web and semantic web query languages: A survey. In Eisinger, N., Maluszynski, J., eds.: Reasoning Web. Volume 3564 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2005) 95
5. Barceló, P., Hurtado, C.A., Libkin, L., Wood, P.T.: Expressive languages for path queries over graph-structured data. In Paredaens, J., Gucht, D.V., eds.: PODS, ACM (2010) 3–14
6. Harrison, J., Théry, L.: A skeptic's approach to combining HOL and Maple. Journal of Automated Reasoning **21** (1998) 279–294
7. Sutcliffe, G., Schulz, S., Claessen, K., Van Gelder, A.: Using the TPTP language for writing derivations and finite interpretations. In Furbach, U., Shankar, N., eds.: Automated Reasoning. Volume 4130 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 67–81
8. Allen, S.F., Bickford, M., Constable, R.L., Eaton, R., Kreitz, C., Lorigo, L., Moran, E.: Innovations in computational type theory using Nuprl. Journal of Applied Logic **4**(4) (December 2006) 428–469
9. Necula, G., Lee, P.: Proof generation in the Touchstone theorem prover. In McAllester, D., ed.: Automated Deduction - CADE-17. Volume 1831 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, Berlin, Heidelberg (2000) 25–44
10. Denney, E.: A prototype proof translator from HOL to Coq. In Aagaard, M., Harrison, J., eds.: TPHOLs. Volume 1869 of Lecture Notes in Computer Science., Springer (2000) 108–125
11. Pons, O., Bertot, Y., Rideau, L.: Notions of dependency in proof assistants. In: Proc. User Interfaces for Theorem Provers, UITP'98. (1998)
12. Urban, J.: MizarMode - an integrated proof assistance tool for the Mizar way of formalizing mathematics. J. Applied Logic **4**(4) (2006) 414–427
13. da Silva, P.P., Sutcliffe, G., Chang, C., Ding, L., Rio, N.D., McGuinness, D.L.: Presenting TSTP proofs with inference web tools. In Konev, B., Schmidt, R.A., Schulz, S., eds.: PAAR/ESHOL. Volume 373 of CEUR Workshop Proceedings., CEUR-WS.org (2008)
14. Dasilva, P., McGuinness, D., Fikes, R.: A proof markup language for semantic web services. Information Systems **31**(4-5) (June 2006) 381–395
15. Buneman, P., Fernandez, M., Suciu, D.: UnQL: a query language and algebra for semistructured data based on structural recursion. The VLDB Journal **9**(1) (March 2000) 76–110
16. Cardelli, L., Gardner, P., Ghelli, G.: A spatial logic for querying graphs. In Widmayer, P., Ruiz, F.T., Bueno, R.M., Hennessy, M., Eidenbenz, S., Conejo, R., eds.: ICALP. Volume 2380 of Lecture Notes in Computer Science., Springer (2002) 597–610
17. Crew, R.F.: ASTLOG: A language for examining abstract syntax trees. In: DSL, USENIX (1997)
18. Jarzabek, S.: Design of flexible static program analyzers with PQL. IEEE Trans. Software Eng. **24**(3) (1998) 197–215